Electronic Communications of the EASST
Volume 64 (2013)

Proceedings of the XIII Spanish Conference on Programming and Computer Languages (PROLE 2013)

R-SQL: An SQL Database System with Extended Recursion[1]

Gabriel Aranda, Susana Nieva, Fernando Sáenz-Pérez and Jaime Sánchez-Hernández

18 pages

Guest Editors: Clara Benac Earle, Laura Castro, Lars-Åke Fredlund
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

[1] This work has been partially supported by the Spanish projects TIN2013-44742-C4-3-R (CAVI-ART), TIN2008-06622-C03-01 (FAST-STAMP), S2009/TIC-1465 (PROMETIDOS), and GPD-UCM-A-910502.

R-SQL: An SQL Database System with Extended Recursion[†]

Gabriel Aranda[1], Susana Nieva[1], Fernando Sáenz-Pérez[2] and Jaime Sánchez-Hernández[1]

[1] Dept. Sistemas Informáticos y Computación, UCM, Spain
[2] Dept. Ingeniería del Software e Inteligencia Artificial, UCM, Spain
garanda@fdi.ucm.es, nieva@sip.ucm.es, fernan@sip.ucm.es, jaime@sip.ucm.es

Abstract:

The relational database language SQL:1999 standard supports recursion, but this approach is limited to the linear case. Moreover, mutual recursion is not supported, and negation cannot be combined with recursion. We designed the language R-SQL to overcome these limitations in [ANSS13], improving termination properties in recursive definitions. In addition we developed a proof of concept implementation of an R-SQL system. In this paper we describe in detail an improved system enhancing performance. It can be integrated into existing RDBMS's, extending them with the aforementioned benefits of R-SQL. The system processes an R-SQL database definition obtaining its extension in tables of an RDBMS (such as PostgreSQL and DB2). It is implemented in SWI-Prolog and it produces a Python script that, upon execution, computes the result of the R-SQL relations. We provide some performance results showing the efficiency gains w.r.t. the previous version. We also include a comparative analysis including some representative relational a deductive systems.

Keywords: Databases, SQL, Recursion, Fixpoint Semantics

1 Introduction

Recursion is a powerful tool nowadays included in almost all programming systems. However, for current implementations of the declarative programming language SQL, this tool is heavily compromised or even not supported at all (MySQL, MS Access, ...) Those systems including recursion suffer from several drawbacks. Linearity is required, so that relation definitions with calls to more than one recursive relation are not allowed. Mutual recursion, and query solving involving an EXCEPT clause are not supported. In general, termination is manually controlled by limiting the number of iterations instead of detecting that there are no further opportunities to develop new tuples. Duplicate discarding is not supported and, so, queries that are actually terminating are not detected as such.

Starburst [MP94] was the first non-commercial RDBMS to implement recursion whereas IBM DB2 was the first commercial one. ANSI/ISO Standard SQL:1999 included for the first time

[†] This work has been partially supported by the Spanish projects TIN2013-44742-C4-3-R (CAVI-ART), TIN2008-06622-C03-01 (FAST-STAMP), S2009/TIC-1465 (PROMETIDOS), and GPD-UCM-A-910502.

recursion in SQL. Today, we can find recursion in several systems: IBM DB2, Oracle, MS SQL Server, HyperSQL and others with the aforementioned limitations.

In [ANSS13] we proposed a new approach, called R-SQL, aimed to overcome these limitations and others, allowing in particular cycles in recursive definitions of graphs and mutually recursive relation definitions. In order to combine recursion and negation, we applied ideas from the deductive database field, such as stratified negation, based on the definition of a dependency graph between the relations involved in the database [Ull89]. We developed a formal framework following the original relational data model [Cod70], therefore avoiding both duplicates and nulls (as encouraged by [Dat09]). We used a stratified fixpoint semantics and we presented an R-SQL database system as a prototype implementing such formal framework. The system can be downloaded from `https://gpd.sip.ucm.es/trac/gpd/wiki/GpdSystems/RSQL`. In this work, we describe in detail an improved version enhancing performance. The system processes an R-SQL database definition obtaining its extension in tables of an RDBMS (such as PostgreSQL and DB2). It is implemented in SWI-Prolog and it generates a Python script that, upon execution, computes the result of the R-SQL relations. The improvements in efficiency relies on a new stratification and a more elaborated version of the fixpoint calculation algorithm that allows to avoid recomputations along iterations. The new system is available at `https://gpd.sip.ucm.es/trac/gpd/wiki/GpdSystems/RSQLplus`. In addition, we experiment with some previously proposed optimizations [Ull89] to improve the performance of the fixpoint computation.

Related academic approaches include DLV^{DB} [TLLP08], LDL++ [AOT+03] (now abandoned and replaced by DeALS, which does not refer to SQL queries up to now), and DES [SP13]. The first one, resulting of a spin-off at Calabria University, is the closer to our work as it produces SQL code to be executed in the external database with a semi-naïve strategy, but lacks formal support for its proposal, and it does not describe non-linear recursion. Last two ones also allow connecting to external databases, but processing of recursive SQL queries are in-memory.

The paper is organized as follows: In Section 2 we recall the syntax and the meaning of R-SQL database definitions. Section 3 describes the system, including the new form of stratification, the fixpoint algorithm, and some performance measurements, showing the efficiency gains for several optimizations. We also include a comparative analysis including some representative relational and deductive systems. Conclusions and future work are summarized in Section 4.

2 Introducing R-SQL

In this section, we present an overview of the language R-SQL, which is focused on the incorporation of recursive relation definitions. The idea is simple and effective: A relation is defined with an assignment operation as a named query (view) that can contain a self reference, i.e., a relation `R` can be defined as `R sch` := `SELECT...FROM...R...`, where `sch` is the relation schema.

2.1 The Definition Language of R-SQL

The formal syntax of R-SQL is defined by the grammar in Figure 1. In this grammar, productions start with lowercase letters whereas terminals start with uppercase (SQL terminal symbols

```
db       ::= R sch := sel_stm; ... R sch := sel_stm;
sch      ::= (A T, ..., A T)
sel_stm  ::= SELECT exp, ..., exp [FROM R, ..., R [WHERE wcond]]
           | sel_stm UNION sel_stm
           | sel_stm EXCEPT R
exp      ::= C | R.A | exp opm exp | − exp
wcond    ::= TRUE | FALSE | exp opc exp | NOT(wcond) | wcond [AND | OR] wcond
opm      ::= + | − | / | *
opc      ::= = | <> | < | > | >= | <=
```

R stands for relation names, A for attribute names, T for standard SQL types and C for constants belonging to a valid SQL type.

Figure 1: A Grammar for the R-SQL Language

use small caps). As usual, optional statements are delimited by square brackets and alternative sentences are separated by pipes.

The language R-SQL overcomes some limitations present in current RDBMS's following SQL:1999. These languages use NOT IN and EXCEPT clauses to deal with negation, and WITH RECURSIVE to engage recursion. As it is pointed out in [GUW09], SQL:1999 does not allow an arbitrary collection of mutually recursive relations to be written in the WITH RECURSIVE clause.

A bundle of R-SQL database examples can be found with the system distribution. Next, we present some of them, to show the expressiveness of the definition language. Each of them is intended to illustrate a concrete aspect of the language in a simple and concise way. Section 3 explores a larger and more natural example.

Mutual Recursion Although any mutual recursion can be converted to direct recursion by inlining [KRP93], our proposal allows to explicitly define mutual recursive relations, which is an advantage in terms of program readability and maintenance. For instance, the following R-SQL database defines the relations `even` and `odd`, as the classical specification of even and odd numbers up to a bound (100 in the example):

```
even(x float) := SELECT 0 UNION SELECT odd.x+1 FROM odd WHERE odd.x<100;

odd(x float) := SELECT even.x+1 FROM even WHERE even.x<100;
```

Nonlinear Recursion The standard SQL restricts the number of allowed recursive calls to be only one. Here we show how to specify Fibonacci numbers in R-SQL[1]:

[1] The relations `fib1` and `fib2` simply represent two aliases for `fib`, which are necessary because, for simplicity, we have not introduced the usual syntax for renamings in the grammar of Figure 1.

```
fib1(n float, f float) := SELECT fib.n, fib.f FROM fib;

fib2(n float, f float) := SELECT fib.n, fib.f FROM fib;

fib(n float, f float) := SELECT 0,1 UNION SELECT 1,1 UNION
  SELECT fib1.n+1,fib1.f+fib2.f FROM fib1,fib2
  WHERE fib1.n=fib2.n+1 AND fib1.n<10;
```

Duplicates and Termination Non termination is another problem that arises associated to recursion when coupled with duplicates. For instance, the following standard SQL query (that considers a finite relation `t`) makes current systems either to reject the query or to go into an infinite loop (some systems allow to impose a maximum number of iterations as a simple termination condition, as DB2):

```
WITH RECURSIVE v(a) AS  SELECT * FROM t UNION ALL SELECT * FROM v
  SELECT * FROM v
```

Nevertheless, the fixpoint computation for the corresponding R-SQL relation:

```
v(a float) := SELECT * FROM t UNION SELECT * FROM v;
```

guarantees termination because duplicates are discarded[2] and `v` does not grow unbounded. The very same termination problem also happens in current RDBMS's with the basic transitive closure over graphs including cycles, but not in R-SQL which ensures termination for finite graphs.

2.2 The meaning of an R-SQL database definition

In [ANSS13] we formalized an operational semantics for the language R-SQL based on stratified negation and fixpoint theory, here we summarize the main ideas.

Stratification is based on the definition of a *dependency graph* $DG_{\mathtt{db}}$ for an R-SQL database `db` that is a directed graph whose nodes are the relation names defined in `db`, and the edges, that can be *negatively labelled*, are determined as follows. A relation definition of the form `R sch` := `sel_stm` in `db` produces edges in the graph from every relation name inside `sel_stm` to `R`. Those edges produced by the relation name that is just to the right of an EXCEPT are negatively labelled.

If there are n relations defined in `db`, and we denote by `RN` the set of the relation names defined in `db`, a *stratification* of `db` is a mapping $str : \mathtt{RN} \to \{1,\ldots,n\}$, such that for every two relations $\mathtt{R}_1, \mathtt{R}_2 \in \mathtt{RN}$ it satisfies:

- $str(\mathtt{R}_1) \leq str(\mathtt{R}_2)$, if there is a path from $\mathtt{R}_1$ to $\mathtt{R}_2$ in $DG_{\mathtt{db}}$,
- $str(\mathtt{R}_1) < str(\mathtt{R}_2)$ if there is a path from $\mathtt{R}_1$ to $\mathtt{R}_2$ in $DG_{\mathtt{db}}$ with at least one negatively labelled edge.

[2] Note that UNION does not require ALL, as current RDBMS's do.

An R-SQL database db is *stratifiable* if there exists a stratification for it. We denote by *numStr* the maximum stratum of the elements of RN.

Intuitively, a relation name preceded by an EXCEPT operator plays the role of a negated predicate (relation) in the deductive database field. A stratification-based solving procedure ensures that when a relation that contains an EXCEPT in its definition is going to be calculated, the meaning of the inner negated relation has been completely evaluated, avoiding nonmonotonicity, as it is widely studied in Datalog [Ull89].

We say that an interpretation I is the relationship between every relation name R and its instance $I(\mathtt{R})$. Interpretations are classified by strata; an interpretation belonging to a stratum i gives meaning to the relations of strata less or equal to i. If I_1, I_2 are two interpretations of stratum i, we say I_1 *is less or equal than* I_2 *at stratum* i, denoted by $I_1 \sqsubseteq_i I_2$, if the following conditions are satisfied for every $\mathtt{R} \in \mathtt{RN}$:

- $I_1(\mathtt{R}) = I_2(\mathtt{R})$, if $str(\mathtt{R}) < i$.
- $I_1(\mathtt{R}) \subseteq I_2(\mathtt{R})$, if $str(\mathtt{R}) = i$.

The meaning of every sel_stm w.r.t. an interpretation I can be understood as the set of tuples (in the current instance represented by I) associated to the corresponding equivalent *RA*-expression, denoted by $[\mathtt{sel\_stm}]^I$. This *RA*-expression is defined as follows: [3]

- $[\text{SELECT } \mathtt{exp}_1,\ldots,\mathtt{exp}_k \text{ FROM } \mathtt{R}_1,\ldots,\mathtt{R}_m \text{ WHERE } \mathtt{wcond}]^I = \pi_{\mathtt{exp}_1,\ldots,\mathtt{exp}_k}(\sigma_{\mathtt{wcond}}(I(\mathtt{R}_1)\times\ldots\times I(\mathtt{R}_m)))$
- $[\mathtt{sel\_stm}_1 \text{ UNION } \mathtt{sel\_stm}_2]^I = [\mathtt{sel\_stm}_1]^I \cup [\mathtt{sel\_stm}_2]^I$
- $[\mathtt{sel\_stm} \text{ EXCEPT } \mathtt{R}]^I = [\mathtt{sel\_stm}]^I - I(\mathtt{R})$

Example 1 Consider the definitions of the relations odd and even of Section 2. Let us assume a concrete interpretation I such that $I(\mathtt{even}) = \{(0),(2)\}$ and $I(\mathtt{odd}) = \emptyset$. Hence, the interpretation of the select statement that defines the relation odd w.r.t. I is:

$[\mathtt{SELECT\ even.x+1\ FROM\ even\ WHERE\ even.x<100}]^I =$
$\{(\mathtt{even.x}+1)[a/\mathtt{even.x}] \mid (a) \in I(\mathtt{even}), (\mathtt{even.x}<100)[a/\mathtt{even.x}] \text{ is satisfied } \} =$
$\{(1),(3)\}$

The case of the relation even is analogous:

$[\mathtt{SELECT\ 0\ UNION\ SELECT\ odd.x+1\ FROM\ odd\ WHERE\ odd.x<100}]^I =$
$[\mathtt{SELECT\ 0}]^I \cup [\mathtt{SELECT\ odd.x+1\ FROM\ odd\ WHERE\ odd.x<100}]^I =$
$\{(0)\} \cup \{(\mathtt{odd.x}+1)[a/\mathtt{odd.x}] \mid (a) \in I(\mathtt{odd}), (\mathtt{odd.x}<100)[a/\mathtt{odd.x}] \text{ is satisfied}\} =$
$\{(0)\}$

Notice that the interpretation $\hat{I}$ defined by:

$$\hat{I}(\mathtt{even}) = \{(0),(2),\ldots,(100)\} \text{ and } \hat{I}(\mathtt{odd}) = \{(1),(3),\ldots,(99)\}$$

[3] Notice that arithmetic expressions are allowed as arguments in *projection* (π) and *select* (σ) operations.

satisfies:

$$\hat{I}(\texttt{even}) = [\texttt{SELECT 0 UNION SELECT odd.x + 1 FROM odd WHERE odd.x < 100}]^{\hat{I}}$$
$$\hat{I}(\texttt{odd}) = [\texttt{SELECT even.x + 1 FROM even WHERE even.x < 100}]^{\hat{I}}$$

So, to give meaning to a database definition, we are interested in an interpretation, called fix, such that for every $\texttt{R} \in \texttt{RN}$, if $\texttt{sel\_stm}$ is the definition of R, then $fix(\texttt{R}) = [\texttt{sel\_stm}]^{fix}$. In the previous example fix will be $\hat{I}$. Since R can occur inside its definition, for every stratum i, the appropriate interpretation fix_i that gives the complete meaning to each relation of stratum i is the least fixpoint of a continuous operator. These fixpoint interpretations are sequentially constructed from stratum 1 to $numStr$. fix represents the fixpoint of the last stratum and provides the semantics for the whole database.

For every i, $1 \leq i \leq numStr$, we define the continuous operator T_i that transforms interpretations belonging to a stratum i as follows:

- $T_i(I)(\texttt{R}) = I(\texttt{R})$, if $str(\texttt{R}) < i$.
- $T_i(I)(\texttt{R}) = [\texttt{sel\_stm}]^I$, if $str(\texttt{R}) = i$ and $\texttt{R sch} := \texttt{sel\_stm}$ is the definition of R in db.
- $T_i(I)(\texttt{R}) = \emptyset$, if $str(\texttt{R}) > i$.

The operator T_1 has a least fixpoint, which is $\bigsqcup_{n\geq 0} T_1^n(\emptyset)$, where $\emptyset(\texttt{R}) = \emptyset$ for every $\texttt{R} \in \texttt{RN}$. We will denote $\bigsqcup_{n\geq 0} T_1^n(\emptyset)$ by fix_1, i.e., fix_1 represents the least fixpoint at stratum 1.

Consider now the sequence $\{T_2^n(fix_1)\}_{n\geq 0}$ of interpretations of stratum 2, greater than fix_1. Using the definition of T_i and the fact that $fix_1(\texttt{R}) = \emptyset$ for every R such that $str(\texttt{R}) \geq 2$, it is easy to prove, by induction on $n \geq 0$, that this sequence is a chain:

$$fix_1 \sqsubseteq_2 T_2(fix_1) \sqsubseteq_2 T_2(T_2(fix_1)) \sqsubseteq_2 \ldots \sqsubseteq_2 T_2^n(fix_1), \ldots$$

$\{T_2^n(fix_1)\}_{n\geq 0}$ is a chain that has as least upper bound, $\bigsqcup_{n\geq 0} T_2^n(fix_1)$, which is the least fixpoint of T_2 containing fix_1. We denote this interpretation by fix_2. By proceeding successively in the same way it is possible to find fix_{numStr}. In [ANSS13] we have proved that fix_{numStr} is the interpretation fix we are looking for, that associates the set of tuples denoted by its definition to every relation of the database .

3 The Improved R-SQL System

Here we present the R-SQL system, which is based on the fixpoint construction of the previous section. We describe its structure, focusing on the improvements that increase the efficiency of the previous prototype, presented in [ANSS13]. These enhances are essentially due to the stratification described in Section 3.1 and in the factoring-out process incorporated in the fixpoint algorithm presented in Section 3.2.

As we show in Figure 2, the system is loaded in SWI-Prolog to process an R-SQL database definition. First, the system parses the input database, then it builds the dependency graph and the stratification if it exists (it raises an error, otherwise); finally, it produces a Python script that will create the SQL database in an RDBMS. After this process, the user can connect to

the RDBMS in order to query or modify the database. Although we are referring to PostgreSQL in the concrete implementation https://gpd.sip.ucm.es/trac/gpd/wiki/GpdSystems/RSQLplus, it can be straightforwardly applied to other systems.

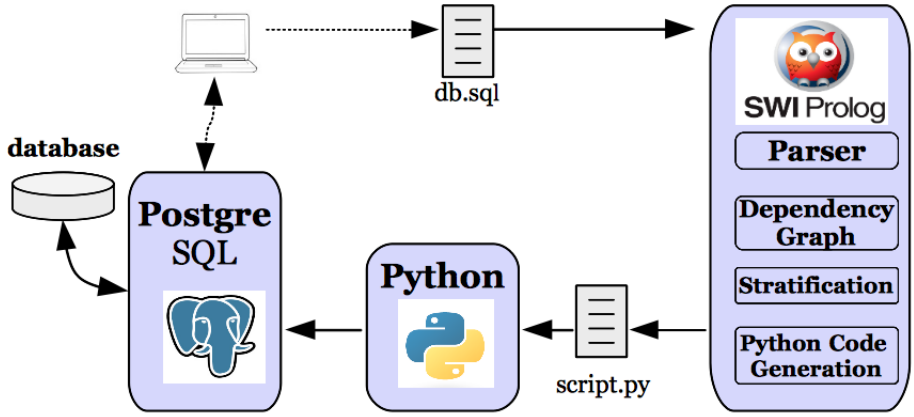

Figure 2: R-SQL System Structure.

Next we present a database for flights to illustrate the process and also will be the working example for the rest of the section. As usual, the information about direct flights can be composed of the city of origin, the city of destination, and the length of the flight. Cities (Lisbon, Madrid, Paris, London, New York) will be represented with constants (`lis`, `mad`, `par`, `lon`, `ny`, resp.). The relation `reachable` consists of all the possible trips between the cities of the database, maybe concatenating more than one flight. The relation `travel` is analogous but also gives time information about alternative trips.

```
flight(frm varchar(10), to varchar(10), time float) :=
  SELECT 'lis','mad',1.0 UNION SELECT 'mad','par',1.5 UNION
  SELECT 'par','lon',2.0 UNION SELECT 'lon','ny',7.0  UNION
  SELECT 'par','ny',8.0;

reachable(frm varchar(10), to varchar(10)) :=
  SELECT flight.frm, flight.to FROM flight UNION
    SELECT reachable.frm, flight.to
    FROM reachable,flight WHERE reachable.to = flight.frm;

travel(frm varchar(10), to varchar(10), time float) :=
  SELECT flight.frm, flight.to, flight.time FROM flight UNION
  SELECT flight.frm, travel.to, flight.time+travel.time
    FROM flight, travel WHERE flight.to = travel.frm;
```

Both `reachable` and `travel` represent transitive closures of the relation `flight`. Notice that if `flight` has a cycle, then the relation `travel` that includes times for each trip is infinite, while `reachable` is not. As pointed before, `reachable` can be finitely computed in our system. But, as `travel` would produce an infinite set of different tuples, some computation limitation would have to be imposed (as the maximum time for a `travel`, for example). However, this is not a drawback of our approach, but an issue due to using infinite relations (built

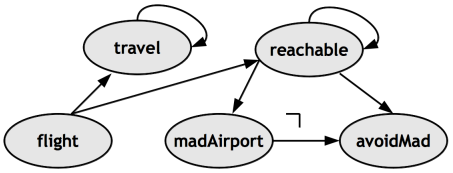

Figure 3: $DG_{\mathtt{db}}$ of the working example.

with arithmetic expressions). The relation `madAirport` contains travels departing or arriving in Madrid, while `avoidMad` contains the possible travels that neither begin, nor end in Madrid.

```
madAirport(frm varchar(10), to varchar(10)) :=
  SELECT reachable.frm, reachable.to FROM reachable
  WHERE (reachable.frm = 'mad' OR reachable.to = 'mad');

avoidMad(frm varchar(10), to varchar(10)) :=
  SELECT reachable.frm, reachable.to FROM reachable EXCEPT madAirport;
```

This definition includes negation together with recursive relations. This combination can not be expressed in SQL:1999 as it is shown in [FMMP96]. The dependency graph of this database is depicted in Figure 3, where negatively labelled edges are annotated with ¬.

3.1 Stratification

Given a database and its dependency graph, there can be a number of different stratifications for it. For instance, for the dependency graph of Figure 4 a possible stratification can assign stratum 1 to the relations $\{\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d},\mathtt{e}\}$ and stratum 2 to $\{\mathtt{f},\mathtt{g}\}$.

For the graph of Figure 4, intuitively it is easy to see that only `b` and `c` must belong to the same stratum due to the mutual dependency between them. The next algorithm minimizes the number of relations in each stratum, which allows to enhance the efficiency of the fixpoint computation as shown in Section 3.2.

- Compute the *strongly connected components* C from DG_{db}. Negative labels are not relevant initially, but once the components are evaluated, it must be checked if there exists some cycle with a negatively labeled edge. In such a case, `db` is not stratifiable and the computation stops. For the example of Figure 4 the components are $\{\mathtt{a}\}$, $\{\mathtt{f}\}$, $\{\mathtt{g}\}$, $\{\mathtt{b},\mathtt{c}\}$, $\{\mathtt{d}\}$ and $\{\mathtt{e}\}$.

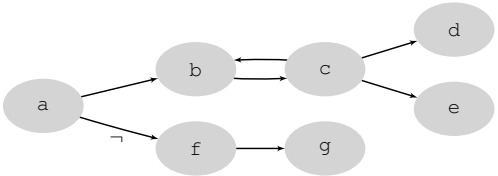

Figure 4: Dependency Graph Example

- *Collapse each strongly connected component* obtaining a new graph with a node for each component, C, and with an edge from C to C' if and only if C contains a relation R and C' contains a relation R′, such that there is an edge from R to R′ in DG_{db}. In our example, the component {b, c} can be collapsed to the node bc, and the rest to its single element. The new graph has the edges {a → bc, bc → d, bc → e, a → f, f → g}.

- Obtain a *topological sorting* for the resulting graph. In our example we can get the sorting a < f < g < bc < e < d.

- Uncollapse the nodes of such a sorting for obtaining a topological sorting for the strongly connected components, and enumerate them in ascending order. In our example, we get {a} < {f} < {g} < {b, c} < {e} < {d}.

  Then, the expected stratification $str(\mathtt{a}) = 1$; $str(\mathtt{f}) = 2$; $str(\mathtt{g}) = 3$; $str(\mathtt{b}) = str(\mathtt{c}) = 4$; $str(\mathtt{e}) = 5$; $str(\mathtt{d}) = 6$ is obtained.

The concrete implementation of this algorithm in R-SQL uses the library *ugraphs* of SWI-Prolog and the module *scc* implemented by Markus Triska, accessible from http://www.logic.at/prolog/scc.pl. For the dependency graph of Figure 3, R-SQL assigns stratum 1 to flight, 2 to travel, 3 to reachable, 4 to madAirport, and 5 to avoidMad.

3.2 The Computation of the Database Fixpoint

Next, we present the algorithm for generating the SQL database corresponding to the fixpoint of an R-SQL database definition db. This algorithm is shown in Figure 5. It produces the SQL statements (CREATE and INSERT) needed to build such a database.

```
 1  for all R ∈ RN_db do
 2    CREATE TABLE R sch;
 3  end for
 4  i := 1
 5  while i ≤ numStr do
 6    for all R ∈ RN_i do
 7      INSERT INTO R out(sel_stm_R);
 8    end for
 9    repeat
10      size := rel_size(RN_i)
11      for all R ∈ RN_i do
12        INSERT INTO R in(sel_stm_R) EXCEPT SELECT * FROM R;
13      end for
14    until size = rel_size(RN_i)
15    i := i + 1
16  end while
```

Figure 5: Algorithm to Compute the Fixpoint

The algorithm considers a concrete stratification for the database where *numStr* denotes the number of strata and $\texttt{NR}_i$ the set of relations of stratum i. First of all, a table is created for each relation `R sch` $:=$ $\texttt{sel\_stm}_\texttt{R}$ of the database (lines 1-3). Then, the external *while* at line 5 computes successively the fixpoints $fix_1, fix_2, \ldots, fix_{numStr}$. Following the semantics, each fix_i is calculated for every relation of $\texttt{NR}_i$, by iterating the fixpoint operators T_i, i.e., the internal *repeat* (lines 9-14) at iteration n computes $T_i^n(fix_{i-1})$. The loop is iterated while some tuple is added to the tables of the current stratum; the variable *size* is used to check this condition.

This algorithm enhances the introduced in [ANSS13] by reducing the work in the iterations of the *repeat*, i.e., simplifying the operations done for filling the tables, so improving the efficiency. The idea is that the iteration of the operator T_i is only needed for recursive relations, and even more precisely, only for the recursive fragment of the select statements defining those relations. With this aim we have defined the functions *in* and *out* to split each `sel_stm` into, respectively, the (recursive) fragment that must be used in the INSERT statements inside the loop, and the fragment that can be processed before the loop, as the base case of the recursive definition. Then, the *for* at lines 6-8 processes the *out* fragments, and the INSERT's at lines 11-13 only process the *in* fragments. The *in* and *out* fragments of a `sel_stm` can be easily determined using the stratum of its components because the stratification defined in the Section 3.1 is such that if a relation `R` in stratum i depends on another relation $\texttt{R}'$, then the stratum of $\texttt{R}'$ is lower than i, so it must be previously computed, or it is exactly i (if they are mutually recursive) and both relations must be computed simultaneously. Therefore, if for instance `R :=` $\texttt{sel\_stm}_1$ UNION $\texttt{sel\_stm}_2$, $str(\texttt{R}) = i$, and $str(\texttt{sel\_stm}_1) < i$, then $\texttt{sel\_stm}_1$ will be part of the *out* fragment, and the corresponding tuples can be inserted before the loop, because the involved relations are already computed in the computation of a previous stratum. Functions *in* and *out* can be easily defined using the stratification as follows:

If $str(\texttt{sel\_stm}) < i$ then we have:

- $in(\texttt{sel\_stm}) = \emptyset$ and $out(\texttt{sel\_stm}) = \texttt{sel\_stm}$.

If $str(\texttt{sel\_stm}) = i$ then, the functions are defined by recursion on the structure of `sel_stm`:

- `sel_stm` $\equiv$ SELECT `exp` ... `exp` FROM `R` ... `R` WHERE `wcond`
  $in(\texttt{sel\_stm}) = \texttt{sel\_stm}$ and $out(\texttt{sel\_stm}) = \emptyset$
- $\texttt{sel\_stm} \equiv \texttt{sel\_stm}_1$ UNION $\texttt{sel\_stm}_2$
  - If $str(\texttt{sel\_stm}_1) = str(\texttt{sel\_stm}_2) = i$ then:
    $in(\texttt{sel\_stm}) = in(\texttt{sel\_stm}_1)$ UNION $in(\texttt{sel\_stm}_2)$ and
    $out(\texttt{sel\_stm}) = out(\texttt{sel\_stm}_1)$ UNION $out(\texttt{sel\_stm}_2)$
  - If $str(\texttt{sel\_stm}_1) = i$ and $str(\texttt{sel\_stm}_2) < i$ then:
    $in(\texttt{sel\_stm}) = in(\texttt{sel\_stm}_1)$ and $out(\texttt{sel\_stm}) = out(\texttt{sel\_stm}_1)$ UNION $\texttt{sel\_stm}_2$
  - If $str(\texttt{sel\_stm}_1) < i$ and $str(\texttt{sel\_stm}_2) = i$ then:
    $in(\texttt{sel\_stm}) = in(\texttt{sel\_stm}_2)$ and $out(\texttt{sel\_stm}) = \texttt{sel\_stm}_1$ UNION $out(\texttt{sel\_stm}_2)$
- $\texttt{sel\_stm} \equiv \texttt{sel\_stm}_1$ EXCEPT $\texttt{sel\_stm}_2$
  $in(\texttt{sel\_stm}) = in(\texttt{sel\_stm}_1)$ EXCEPT $\texttt{sel\_stm}_2$ and
  $out(\texttt{sel\_stm}) = out(\texttt{sel\_stm}_1)$ EXCEPT $\texttt{sel\_stm}_2$

The concrete implementation of the algorithm of Figure 5 can be done in a number of ways. We have chosen Python as the host language mainly because it is multiplatform and provides easy connections with different database systems such as PostgreSQL, DB2, MySQL, or even via ODBC, which allows connectivity to almost any RDBMS. The additional features required for the host language are basic: Loops, assignment and simple arithmetic.

Below, we show the Python code generated for the working example of flights. It uses the Python library *psycopg2* (available at http://initd.org/psycopg/) which allows to connect to an RDBMS and then submit SQL queries as:

```
cursor.execute("<query>")
```

where <query> is any valid SQL query. The generated code expands all the loops of the algorithm of Figure 5, except the *repeat* at lines 9-14. As Python does not provide a *repeat* (or *do-while*) loop construction, we implement it as a *while True* sentence with the corresponding *break* for stopping it when the condition holds. We will show it in the code generated for stratum 2. Moreover, we also implement a Python function relSize(<list of relations>) that returns the number of tuples of the relations specified in its argument.

The *for* at lines 1-3 is expanded as:

```
cursor.execute("CREATE table flight
                    (frm varchar(10), to varchar(10), time float);")
cursor.execute("CREATE table travel
                    (frm varchar(10), to varchar(10), time float);")
```

and so on for the rest of relations. Now, we detail some parts of the code generated stratum by stratum. For stratum 1 the *in* fragment is empty and we have:

```
# Code generated for Stratum 1
cursor.execute("INSERT INTO flight
           (SELECT 'lis','mad',1 UNION SELECT 'mad','par',1.5 UNION
            SELECT 'par','lon',2 UNION SELECT 'lon','ny',7 UNION
            SELECT 'par','ny',8) EXCEPT
           SELECT * FROM flight;")
```

Stratum 2 contains the relation travel whose definition can be splitted into two parts with the functions *in* and *out*.

```
# Code generated for Stratum 2
# out fragment
cursor.execute("INSERT INTO travel (SELECT * FROM flight);")

# in fragment
while True:
  cursor.execute("INSERT INTO travel
              (SELECT flight.frm,travel.to,flight.time+travel.time
               FROM flight,travel WHERE flight.to = travel.frm)
               EXCEPT SELECT * FROM travel;")

  newSize = relSize(["travel"])

  if (newSize != size):
       size = newSize
  else:
       break
```

The tuples added for `travel` at each iteration of this code are shown in the next Table:

	Set of added tuples
out fragment	{(lon,ny,7.0),(par,lon,2.0),(par,ny,8.0), (mad,par,1.5),(lis,mad,1.0)}
in fragment: iteration 1	{(lis,par,2.5),(par,ny,9.0),(mad,ny,9.5),(mad,lon,3.5)}
in fragment: iteration 2	{(lis,ny,10.5),(lis,lon,4.5),(mad,ny,10.5)}
in fragment: iteration 3	{(lis,lon,4.5),(mad,ny,10.5),(lis,ny,11.5)}

Analogously, the system produces the Python code for strata 3 and 4, which correspond to `reachable` and `madAirport`, respectively. Finally, in the last stratum the `avoidMad` relation is computed (there is no *in* fragment in this case):

```
# Code generated for Stratum 5
# out fragment
cursor.execute("INSERT INTO avoidMad
            (SELECT travel.frm,travel.to FROM travel
             EXCEPT SELECT * FROM madAirport)");
```

This completes the fixpoint for the working example database. The values for `flight`, `madAirport` and `avoidMad` tables are illustrated in the graph in Figure 6. Direct flights are represented in blue color and labeled with their corresponding time. Paths for `madAirport` relation are represented in red color and path for `avoidMad` relation are represented in black color.

Once the R-SQL database definition has been processed, the tables obtained are available as a database instance in PostgreSQL. Then, the user can formulate queries that will be solved using those tables (without performing any further fixpoint computation).

3.3 Performance

This section analyzes the system performance. First, we focus on the improvement of factoring out SQL fragments (as already explained in Section 3.2). And, second, we develop a field analysis by targeting the system to different current state-of-the art relational systems, introducing the

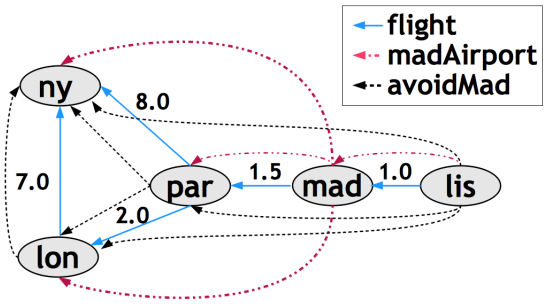

Figure 6: Graphical representation of resulting values of the working example.

benefits of a semi-naïve differential optimization [Ull85] for linear recursive queries. Numbers for tables in this section are expressed in milliseconds and represent the average of a number of runs, where the maximum and minimum have been elided.

3.3.1 Factoring-Out Improvement

As introduced, any DBMS allowing Python access can be used to implement our proposal. This section develops the connection to IBM DB2 as a target system for analyzing the performance. We consider the benchmark `reachable` that implements the transitive closure of the relation `flight`, as introduced in Section 3. To build a parametric relation, we consider links in flights as the tuples $\{(1,2),(2,3),\ldots,(n,n+1)\}$, where $n+1$ is the number of nodes in the graph and the type of the fields have been changed to integer. Table 1 shows the results for instances of this benchmark with a number of tuples ranging from 100 to 350 (first column). Second column lists the number of tuples generated in the result set. Third and fourth columns show the elapsed running time for solving the query in R-SQL with no factoring-out improvement (No FOI) and with this improvement enabled (With FOI), respectively. Fifth column (Speed-up) shows the speed-up due to FOI as a percentage. The last column (Difference) shows the absolute time difference between both timings. Benchmarks have been run on an Intel Core2 Quad CPU at 2.4GHz and 3GB RAM, running Windows XP 32bit SP3, and IBM DB2 Express Edition 10.1.0 database server with a default configuration.

Tuples	Result Tuples	No FOI	With FOI	Speed-up	Difference
100	5,050	1,135	1,050	8.1%	85
150	11,325	4,438	3,428	29.4%	1,010
200	20,100	10,048	8,172	23.0%	1,876
250	31,375	19,001	16,041	18.5%	2,960
300	45,150	32,710	28,381	15.3%	4,329
350	61,425	50,085	44,175	13.4%	5,910

Table 1: Factoring-Out Improvement (FOI)

From this experiment we confirm the expected results for factoring the fragment `select * from flight` out of the recursive clause and the *repeat* loop. Indeed, even for a single SQL fragment as this, speed-ups of up to almost 30% are reached. However, as long as the tuples do increase in the instances, the speed-up decrease because the main computation effort corresponds to the *repeat* loop because of the EXCEPT operator.

Next section deals with other optimizations and comparison with other relational and deductive systems.

3.3.2 Analysis of Systems

This section considers different current state-of-the-art relational systems which include recursive queries: PostgreSQL 9.3, Oracle 11g, and DB2 10.1, all of them with a default configuration. We compare R-SQL when solving the previous benchmark with these systems and show the

importance of introducing the semi-naïve differential optimization [Ull85]. To make the comparison fairest with the RDBMS's, which do not discard duplicates, we omit the operator EXCEPT to behave similarly to the optimized R-SQL systems. Also, we include the last published version of DLV^{DB} in this comparison as a deductive system which is able to project its solving to these external databases when computing a transitive closure. Another related deductive system is LDL++, but unfortunately it is not included in this comparison since it has been replaced by the system DeALS whose binaries and/or sources are not available yet.

RDBMS	System	100	200	300	400	500
PostgreSQL	*Native SQL*	161	187	240	360	713
	R-SQL	500	3,198	12,406	39,802	71,922
	Diff-R-SQL	208	459	1,073	2,271	4,115
	TDiff-R-SQL	260	578	1,323	2,745	5,693
	DLV^{DB}	703	1,651	4,458	8,047	13,120
Oracle	*Native SQL*	604	1,781	5,765	13,349	26,297
	R-SQL	880	3,802	12,057	27,989	56,641
	Diff-R-SQL	708	1,437	3,224	6,240	11,469
	TDiff-R-SQL	646	995	1,708	2,453	3,422
	DLV^{DB}	6,875	12,849	18,912	30,583	42,146
DB2	*Native SQL*	677	1,016	1,323	2,052	3,099
	R-SQL	1,271	5,797	97,052	129,917	150,104
	Diff-R-SQL	698	932	2,672	2,859	3,213
	TDiff-R-SQL	646	1,000	1,578	4,021	9,021
	DLV^{DB}	6,339	12,666	53,552	57,349	100,391

Table 2: Analysis of Systems

The results for different instances of the benchmark are given in Table 2. Numbers are now arranged with the parameter n ranging in the horizontal axis, and rows include the considered RDBMS (first column), the system connected to this relational database (second column), and then (in the next five columns), the wall time for solving each instance (from 100 up to 500 tuples in the relation `flight`, which delivers from 5,050 up to 125,250 tuples in the result set of the query benchmark). Below the headings, lines are arranged in major rows, each one referring to a concrete RDBMS. And, for each RDBMS (*PostgreSQL*, *Oracle*, *DB2*)[4], five minor rows are listed, which refer to each system. The first minor row *Native SQL* refers to the corresponding RDBMS, which is used to compare how the rest of the systems behave w.r.t. a native execution of the benchmark, i.e., resorting to the recursive query specification for the transitive closure that each RDBMS provides. For instance, DB2 uses the following syntax (where `rec` is the temporary recursive relation which is built to fill the relation `reachable`):

```
INSERT INTO reachable
  WITH rec(frm,to) AS
```

[4] Incidentally, MySQL does not support recursive queries at all.

```
    (SELECT * FROM flight
     UNION ALL
     SELECT flight.frm, rec.to FROM flight,rec
     WHERE flight.to = rec.frm)
   SELECT * FROM rec;
```

The next minor row *R-SQL* refers to the implementation we have presented in Section 3. Minor row labeled with *Diff-R-SQL* presents the results for R-SQL with the semi-naïve differential optimization enabled as explained in [Ull85]. Roughly, for a linear query, this optimization refers to use in each iteration only the results that have been generated in the previous iteration to build new tuples. To implement this, we have resorted to add a new integer column (`it` in the benchmark) holding the iteration in which a given tuple has been generated. For instance, the next query is executed for each iteration `$IT$` (this is substituted by the actual iteration number along iterations):

```
INSERT INTO reachable
  SELECT flight.frm, reachable.to, $IT$
  FROM flight, reachable
  WHERE flight.to = reachable.frm AND
        reachable.it = $IT$-1;
```

Next, the row labeled with *TDiff-R-SQL* refers to an alternative implementation of the semi-naïve differential optimization, which consists on storing all the tuples generated in a given iteration in a temporary table. Then, the join at each iteration is computed between `flight` and this temporary table, therefore avoiding to scan the growing relation `reachable` looking for the tuples with a given iteration number value in the extra field. In fact, two temporary tables are needed: One for accessing the tuples generated in the previous iteration, and another one to store the new tuples. Next, there is a sketch of the SQL statements submitted in each iteration to DB2, where `reachable_temp1` is intended to hold the tuples generated in the previous iteration, and `reachable_temp2` is for the current one (temporary tables are preceded by `SESSION.`):

```
INSERT INTO SESSION.reachable_temp2
  SELECT flight.ori, SESSION.reachable_temp1.des
  FROM flight, SESSION.reachable_temp1
  WHERE flight.des = SESSION.reachable_temp1.ori;
...
INSERT INTO reachable SELECT * FROM SESSION.reachable_temp1;
DELETE FROM SESSION.reachable_temp1;
INSERT INTO SESSION.reachable_temp1
  SELECT * FROM SESSION.reachable_temp2;
DELETE FROM SESSION.reachable_temp2;
```

The first SQL sentence loads into `reachable_temp2` the results just computed for the current iteration. Next sentences simply load on `reachable` the results from the previous iteration, and transfer the results just available in `reachable_temp2` to `reachable_temp1` in order for them to be available for the next iteration. `reachable_temp2` is finally flushed to be ready for the next iteration as well.

Using temporary tables should reveal an advantage as neither log records nor lock management are needed. They are computed in-memory as much as possible; only when they do not fit into RAM, memory space quota is requested for them.

Finally, the row labeled with DLV^{DB} stands for this deductive system, which uses the same ODBC bridge to access those relational systems.

Looking at the numbers, it is noticeable that the best performance is achieved by the native SQL execution in PostgreSQL for all the considered instances ($n \in \{100, 200, \ldots, 500\}$) of the benchmark. Also, the worst performance corresponds to R-SQL without optimizations (and including the operator EXCEPT), which is also clear as the join and the difference must be processed in each iteration for all the tuples, including those that definitely will not be involved in generating a new one. The semi-naïve differential optimization (which also avoids the operator EXCEPT) alleviates this enormously, with a huge factor of $150,104/3,213 = 46.7\times$, when comparing *R-SQL* vs. *Diff-R-SQL* for DB2. DLV^{DB} is the next system in the performance ranking, behaving better than R-SQL but worse than the rest. Depending on the RDBMS, the next best system can be either *Diff-R-SQL* or *TDiff-R-SQL*: The first one performs better than the second for PostgreSQL and the other way round for Oracle and DB2. Noticeably, both perform better than *Native SQL* for Oracle, and *Diff-R-SQL* behaves roughly similar to DB2. These numbers highlight how similar techniques are differently managed by the different RDBMS's. For example, whereas for Oracle the use of temporary tables is of paramount importance for lowering the solving time, its effect is the contrary for DB2. We have also tested table functions, which provide a way to implement parametric views. However, they do not provide better performance than the already illustrated optimizations (and Oracle faces the mutating table problem when using them to insert tuples in the same source table).

All in all, in the best case we are able to beat an RDBMS by a factor of $26,297/3,422 = 7.7\times$, and in the worst case (but considering the best optimization) we are beaten by a factor of $4,115/713 = 5.8\times$. To better understand this slowdown, we must consider that the R-SQL system runs an interpreted script (Python) and in each iteration, one or several SQL statements are sent to the RDBMS via the ODBC bridge. SQL statements sent in this way must be compiled by the RDBMS for each iteration, so that it becomes a significant burden on the system, together with the communication cost due to the bridge. Therefore, using a compiled language supporting prepared SQL statements should be a point worth to explore for performance gains.

4 Conclusions

R-SQL has been designed to compute the meaning of a database definition and then to query this database. Notice that the modification of a relation of the database in the underlying RDBMS can cause inconsistencies since the tables are not recomputed. For instance, after processing the database for flights, if the user adds or deletes a tuple for the relation `flight`, then the relation `travel` will become inconsistent according to its R-SQL definition. But this is the very same behavior of RDBMS's when dealing with materialized views. A future direction in order to fully integrate R-SQL into an RDBMS is to have the possibility of restoring the consistence of the database (using triggers for instance), as well as to define additional (possibly recursive) views. This restoring involves the recomputation of the database fixpoint. But, using the dependency graph, it is easy to determine the subset of relations that must be calculated, instead of computing the whole fixpoint for the database. Moreover, those relations may not need to be recomputed from scratch. In addition, it is straightforward to modify the algorithm introduced in Section

3.2 to get a lazy evaluation of such relations, performing iterations only when new values are demanded.

As shown in Section 3.3.2, the semi-naïve differential optimization [Ull89] for linear recursive queries has a notable impact on performance. Nonetheless, our system can be further extended for non linear recursive queries and with enhancements as in [ZCF+97, BR87], as DLV [TLLP08] does. Implementing all these optimizations are left for future work.

Although our proposal is encouraging as results reveal, efficiency can also be improved by indexing (e.g., tries [SW12] and BDD's [WACL05]) temporary relations during fixpoint computations. To seamlessly integrate this into an RDBMS, we can profit from the fourth-generation languages (e.g., SQL PL in IBM DB2 and PL/SQL in Oracle) and completely integrate query solving and view maintenance into the RDBMS. This way, prepared SQL statements are available in a compiled setting, which should also improve performance. We are currently extending the R-SQL system with the enhancements aforementioned and more features as hypothetical definitions and aggregates.

Bibliography

[ANSS13] G. Aranda-López, S. Nieva, F. Sáenz-Pérez, J. Sánchez-Hernández. Formalizing a Broader Recursion Coverage in SQL. In *Symposium on Practical Aspects of Declarative Languages (PADL'13)*. LNCS 7752, pp. 93 – 108. 2013.

[AOT+03] F. Arni, K. Ong, S. Tsur, H. Wang, C. Zaniolo. The Deductive Database System LDL++. *TPLP* 3(1):61–94, 2003.

[BR87] I. Balbin, K. Ramamohanarao. A Generalization of the Differential Approach to Recursive Query Evaluation. *J. Log. Program.* 4(3):259–262, 1987.

[Cod70] E. Codd. A Relational Model for Large Shared Databanks. *Communications of the ACM* 13(6):377–390, June 1970.

[Dat09] C. J. Date. *SQL and relational theory: how to write accurate SQL code*. O'Reilly, Sebastopol, CA, 2009.

[FMMP96] S. J. Finkelstein, N. Mattos, I. S. Mumick, H. Pirahesh. Expressing Recursive Queries in SQL. Technical report, ISO, 1996.

[GUW09] H. Garcia-Molina, J. D. Ullman, J. Widom. *Database systems - the complete book (2. ed.)*. Pearson Education, 2009.

[KRP93] O. Kaser, C. R. Ramakrishnan, S. Pawagi. On the conversion of indirect to direct recursion. *ACM Lett. Program. Lang. Syst.* 2(1-4):151–164, Mar. 1993.

[MP94] I. S. Mumick, H. Pirahesh. Implementation of magic-sets in a relational database system. *SIGMOD Rec.* 23:103–114, May 1994.

[SP13] F. Sáenz-Pérez. Towards Bridging the Expressiveness Gap Between Relational and Deductive Databases. In *XIII Jornadas sobre Programación y Lenguajes, PROLE2013 (SISTEDES)*. September 2013.

[SW12] T. Swift, D. S. Warren. XSB: Extending Prolog with Tabled Logic Programming. *TPLP* 12(1-2):157–187, 2012.

[TLLP08] G. Terracina, N. Leone, V. Lio, C. Panetta. Experimenting with recursive queries in database and logic programming systems. *TPLP* 8(2):129–165, 2008.

[Ull85] J. D. Ullman. Implementation of Logical Query Languages for Databases. *ACM Trans. Database Syst.* 10(3):289–321, 1985.

[Ull89] J. Ullman. *Principles of Database and Knowledge-Base Systems Vols. I (Classical Database Systems) and II (The New Technologies)*. Computer Science Press, 1989.

[WACL05] J. Whaley, D. Avots, M. Carbin, M. S. Lam. Using Datalog with binary decision diagrams for program analysis. In *In Proceedings of Programming Languages and Systems: Third Asian Symposium*. 2005.

[ZCF+97] C. Zaniolo, S. Ceri, C. Faloutsos, R. T. Snodgrass, V. S. Subrahmanian, R. Zicari. *Advanced Database Systems*. Morgan Kaufmann Publishers Inc., 1997.